IFT 2015 : Devoir 4

Professeure : Esma Aïmeur
Démonstrateurs : Dorsaf Sallami et Hugo Rocha

Hiver 2024

1 Consignes générales

- Date de rendu : jeudi 25 avril à 23h59.
- Ce travail est à faire seul(e) ou à deux.
- Aucun plagiat ne sera accepté.
- Il est possible que des précisions ou des modifications de l'énoncé soient envoyées par courriel et mises sur Studium.
- Chaque jour de retard pour la remise de votre travail entraînera une pénalité de 5 points.
- Tout code est à faire avec Java JDK 17
- Vous devez soumettre les fichiers Java demandés et un unique PDF avec les réponses écrites.
- Le PDF peut être des photos/scans de réponses écrites mains. Attention à la lisibilité!
- **Si vous êtes en équipe :** l'un soumet les fichiers demandés et l'autre soumet uniquement un PDF avec le nom de son partenaire.
- L'évaluation du code peut être automatisée, assurez-vous d'avoir **les même signatures de fonctions**.
- Vous avez le droit de créer autant de fonctions intermédiaires que vous voulez.
- **N'oubliez pas d'ajouter des docstrings et des commentaires à vos implémentations. Suivez également les principes de CamelCase. Le code non documenté sera pénalisé.**
- Pour toute question sur le devoir 4, nous vous demandons de la publier sur le forum "TP4 Q&A" sur Studium.
- Bon travail!

2 Dictionnaire basé sur un arbre AVL (5 points)

Context: Dans cet exercice, vous implémenterez un dictionnaire en utilisant un Arbre AVL. Le dictionnaire stockera des mots et leurs significations sous forme de paires clé-valeur, chaque mot agissant comme une clé unique (**les clés dupliquées ne sont pas autorisées**).
L'attribut `word` de chaque `AVLNode` agit comme la clé. Il est utilisé pour comparer les nœuds. L'arbre AVL est ordonné en fonction de **l'ordre alphabétique** des clés.
La Classe `AVLNode` pour représenter les nœuds dans l'Arbre AVL. Chaque nœud doit stocker :

- Le mot (clé) comme une chaîne de caractères (`String`).
- La signification (valeur) comme une chaîne de caractères (`String`).
- La hauteur du nœud à des fins d'équilibrage.
- Des pointeurs vers les nœuds enfants gauche et droit.

Travail à faire: Complétez la classe `DictionaryAVLTree` (dans le fichier `DictionaryAVLTree.java`) avec les méthodes suivantes :

- (2pts) `private AVLNode insert(AVLNode node, String word, String meaning)` permet d'ajouter de nouveaux mots et leurs significations au dictionnaire. L'arbre doit rester équilibré après chaque insertion.
- (2pts) `private AVLNode delete(AVLNode root, String word)` permet de retirer des mots du dictionnaire. L'arbre doit rester équilibré après chaque suppression.
- (1pt) `public String search(AVLNode root, String word)` permet de trouver la signification d'un mot donné.

Livrable: Il faut retouner le ficher `DictionaryAVLTree.java`

3 Modélisation d'un Réseau de Métro (15 points)

Contexte: Vous êtes un développeur travaillant pour la Société de transport de Montréal. Votre tâche est de créer une structure de données pour gérer le réseau de métro de la ville, qui permettra de planifier les itinéraires, de mettre à jour les services et d'analyser le trafic.

Dans cet exercice, vous allez travailler avec une structure de données pour un réseau de métro, en utilisant **un graphe** où chaque station est connectée par des itinéraires. Ce graphe sera spécifiquement implémenté en utilisant **des listes doublement chaînées** pour gérer les connexions entre les stations de manière efficace.

Détails de la mise en œuvre:

Pour ce faire, vous utiliserez l'objet `Vertex` pour représenter les stations. Chaque sommet (station) du graphe possède plusieurs attributs essentiels qui facilitent la modélisation du réseau. Dans le fichier `Vertex.java`, chaque sommet du graphe est défini avec les attributs suivants :

- **data (`private E data`):** Type générique `E` représentant les données stockées dans le sommet, comme le nom de la station de métro.
- **inEdges, outEdges (`private DoublyLinkedList<Edge<E,T>>`):** Deux listes doublement chaînées stockant respectivement les arêtes entrantes et sortantes, représentant les lignes de métro arrivant à et partant de la station.
- **position (`private Node<Vertex<E,T>>`):** Référence à la position du sommet dans la structure de données globale, facilitant son accès rapide.
- **status (`private int status`):** Entier indiquant l'état de visite du sommet, utile dans les algorithmes de parcours de graphe.

- **color (`private int color`):** Utilisé dans les algorithmes nécessitant une coloration des sommets, par exemple pour identifier des composantes ou des conditions particulières.
- **id (`private final int id`):** Identifiant unique pour le sommet, permettant de le distinguer des autres sommets.

La classe `Edge` est utilisée pour représenter les connexions, ou les routes, entre les stations de métro. Chaque arête (ou trajet) du graphe possède les attributs suivants, essentiels pour décrire les caractéristiques et les relations entre les stations :

- **v1, v2 (`private Vertex<E,T> v1, v2`):** Représentent les stations (sommets) que cette arête (route) connecte.
- **incidentPositionV1, incidentPositionV2 (`private Node<Edge<E,T>> incidentPositionV1, incidentPositionV2`):** Pointe vers la position de cette arête dans les listes d'arêtes sortantes et entrantes des sommets `v1` et `v2`. Cela facilite la gestion des connexions entre stations.
- **label (`private T label`):** Peut être utilisé pour stocker des informations sur l'arête, telles que le nom de la ligne de métro.
- **weight (`private double weight`):** Un poids associé à l'arête.
- **position (`private Node<Edge<E,T>> position`):** Référence à la position de cette arête dans une structure de données globale.
- **status (`private int status`):** Utilisé pour marquer l'état de l'arête dans le cadre de parcours de graphes ou d'algorithmes de recherche.

Travail à faire:

1. (1pt) Complétez le fichier `Edge.java` en implémentant la fonction `getOpposite(Vertex<E,T> v)`. La méthode doit prendre un sommet v de type `Vertex<E,T>` en paramètre et déterminer quel sommet est à l'extrémité opposée de l'arête par rapport à v.
2. (1pt) Complétez le fichier `Vertex.java` en implémentant la fonction `public Vertex<E,T>[] getNeighbors()`. La méthode doit retourner un tableau contenant tous les sommets voisins du sommet courant. Un sommet voisin est défini comme étant à l'autre extrémité d'une arête partant du sommet courant (dans le cas d'un graphe direct) ou connecté par une arête au sommet courant (dans le cas d'un graphe non direct). Hint: Utilsez la fonction `public Vertex<E,T> getOpposite(Vertex<E,T> v)`

La classe `Graph` est au cœur de la modélisation de votre réseau de métro, fournissant la structure principale pour représenter l'ensemble du système. Voici les attributs et options essentiels de la classe :

- **vertexList (`private DoublyLinkedList<Vertex <E,T>> vertexList`):** Une liste doublement chaînée de tous les sommets (stations) dans le graphe. Chaque élément de la liste représente une station unique dans le réseau de métro.
- **edgeList (`private DoublyLinkedList<Edge<E,T>> edgeList`):** Une liste doublement chaînée contenant toutes les arêtes (routes) du graphe. Chaque élément de cette liste représente une route directe entre deux stations.
- **directed (`private boolean directed`):** Un booléen indiquant si le graphe est dirigé (true) ou non-dirigé (false). Dans un contexte de métro, un graphe dirigé pourrait indiquer des voies unidirectionnelles.
- **isCyclic (`private boolean isCyclic`):** Un booléen qui détermine si le graphe contient des cycles. Cela peut être pertinent pour identifier des boucles dans le réseau de métro.
- **isConnected (`private boolean isConnected`):** Indique si le graphe est connecté, c'est-à-dire si toutes les stations sont accessibles entre-elles par des routes de métro.

- **connectedComponents (`private int connectedComponents`):** Le nombre de composantes connectées dans le graphe. Cela peut aider à comprendre combien de zones distinctes ou non reliées existent dans le réseau de métro.

Travail à faire:

1. (10 pts: 1*10)Complétez le fichier `Graph.java` en implémentant les fonctions suivantes:

Méthode	**Retour**	**Explication**
addEdge	Arêtes ajoutées	Ajouter une arête entre deux sommets. Ajouter une autre dans la direction opposée si le graphe est non orienté
removeEdge	void	Supprime une arête du graphe
removeVertex	void	Supprime un sommet du graphe
areAdjacent	Booléen	Vérifie si deux sommets sont adjacents
BFS	Tableau de sommets	Parcourir le graphe avec la recherche en largeur
DFS	Tableau de sommets	Parcourir le graphe avec la recherche en profondeur
connectedComponents	Nombre de composants connectés	Vérifie combien de composants connectés contient le graphe
isConnected	Booléen	Vérifie si le graphe est connecté
isCyclic	Booléen	Vérifie si le graphe contient des cycles
isDirected	Booléen	Vérifie si le graphe est orienté

2. (1.75pts: 0.25*7)Déterminez la complexité des méthodes en remplissant le tableau ci-dessous.

Méthode	**Complexité**
addEdge	
removeVertex	
areAdjacent	
connectedComponents	
isConnected	
isCyclic	
isDirected	

3. (1.25pt) Exécutez le fichier `GraphMain.java` en utilisant le fichier `STM.txt` (qui contient les stations du STM) et retournez le fichier `Output.txt` résultant.

Livrable: Il faut retouner tous les fichiers dans le dossier graph avec le ficher `Output.txt`